



PROYECTO DE SISTEMAS INFORMÁTICOS

CURSO 2011/2012

# JLOP

(JSON LANGUAGE ORIENTED PROCESSING)

## **Autores**

Javier Peñas Jaramillo

Laura Reyero Sainz

## **Director**

Antonio Sarasa Cabezuelo



Se autoriza a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la memoria como el código, la documentación y/o el prototipo desarrollado.

Javier Peñas Jaramillo

Laura Reyero Sainz



# ÍNDICE

Introducción.....	- 7 -
Estado del arte .....	- 11 -
Procesadores.....	- 11 -
Gramáticas de atributos .....	- 12 -
CUP .....	- 13 -
Estructura de un fichero CUP .....	- 14 -
JAVACC.....	- 17 -
Lenguajes de Mercado .....	- 19 -
Mercado de presentación.....	- 19 -
Mercado de procedimientos .....	- 19 -
Mercado descriptivo .....	- 19 -
JSON .....	- 20 -
Documento JSON .....	- 20 -
Comparación de JSON con XML y otros lenguajes de marcado.....	- 21 -
Objetivo del Proyecto .....	- 23 -
Metagramática JLOP .....	- 25 -
Modelo de proceso .....	- 29 -
Ciclo del proyecto.....	- 29 -
Arquitectura del sistema .....	- 31 -
Caso de estudio.....	- 43 -
Coordenadas geográficas.....	- 49 -
Localización geográfica de un punto.....	- 51 -
Meridianos .....	- 52 -
Paralelos.....	- 53 -
Longitud .....	- 54 -
Latitud.....	- 55 -
Google Maps .....	- 57 -
Conclusiones .....	- 59 -
Bibliografía .....	- 61 -
Anexo 1. Ejemplo completo .....	- 63 -
Anexo 2. Manual de usuario .....	- 65 -
Glosario de términos y acrónimos .....	- 69 -



## INTRODUCCIÓN

El problema abordado en este proyecto de Sistemas Informáticos es el del procesamiento de documentos JSON [2].

JSON [1] es un formato ligero para intercambio de datos que surge como alternativa a XML en AJAX. Se emplea habitualmente en entornos donde el tamaño del flujo de datos entre cliente y servidor es de vital importancia, cuando la fuente de datos es explícitamente de confianza y donde no es importante el no disponer de procesamiento XSLT para manipular los datos en el cliente.

Este formato de datos es más liviano que XML y es en apariencia más sencillo, ya que utiliza el código JavaScript como modelo de datos.

Tiene un formato sencillo y fácil de utilizar tanto para programadores para su lectura y escritura como para máquinas en su análisis y generación.

El objetivo de este proyecto será poder generar un programa que procese documentos JSON de una manera determinada, aplicando una o varias funciones al mismo, que deberán ser previamente especificadas.

**Palabras clave:** JSON, JavaCC, CUP, Procesadores del lenguaje, mapa, coordenadas geográficas, gramáticas de atributos.





# Introduction

The problem addressed in this project of Sistemas Informáticos is about JSON [1] document processing.

JSON [1] is a lightweight format for data exchange that emerges as an alternative to XML in AJAX. Is typically used in environments where the size of the data stream between client and server is vital, when the data source is explicitly trusted and is not important the absence of XSLT processing to manipulate the data on the client.

This data format is lighter than XML and is apparently easier because it uses JavaScript as a data model.

JSON has a simple format and it's easy to use for both programmers to read and write and for machines in its analysis and generation.

The aim of this project is to create a program to process JSON documents in a certain way, by one or more functions to it, which must be previously specified.

**Key words:** JSON, JavaCC, CUP, Lenguaje processors, map, geographic coordinates, attribute grammar



# ESTADO DEL ARTE

## PROCESADORES

Un procesador de un lenguaje es una aplicación cuya entrada es un lenguaje y se encarga de hacer cualquier tipo de procesamiento (reconocimiento, aceptación, procesado...) del mismo.

Las principales tareas de un procesador de lenguajes [29] se pueden dividir en los siguientes apartados:

### 1) Análisis

- a. Análisis Léxico del lenguaje origen. Primera fase de un compilador que recibe como entrada el código fuente de otro programa (secuencia de caracteres) y produce una salida compuesta de tokens (componentes léxicos) o símbolos. Estos tokens sirven para una posterior etapa del proceso de traducción, siendo la entrada para el analizador sintáctico.
- b. Análisis Sintáctico. Con los tokens generados en la fase anterior, se construye una estructura de datos, por ejemplo un árbol de análisis o árboles de sintaxis abstracta. Mediante una gramática como puede ser EBNF se indica la sintaxis de las asignaciones, inicio y fin del programa etc.
- c. Análisis semántico. Revisa el programa fuente para tratar de encontrar errores semánticos y reúne la información sobre los tipos para la fase posterior de generación de código. En ella se utiliza la estructura jerárquica determinada por la fase de análisis sintáctico para identificar los operadores y operandos de expresiones y proposiciones.

### 2) Síntesis

- a) Generación del código intermedio. Una vez pasada la fase de análisis se genera el código intermedio (bytecode) común para todas las máquinas.
  - b) Generación del código objeto. A partir de este código intermedio se genera el código objeto para poder ejecutar el programa. En algunos casos lo que se hace es generar un fichero objeto en código ensamblador y a partir de este código se genera el ejecutable ".exe".
- 3) Tablas de símbolos y control de errores, que no entran dentro de las fases de análisis. En todas las tareas sean de análisis o síntesis es necesario poder utilizar la pila de errores para registrar cualquier error. También es necesario la tabla de símbolos para la traducción de un lenguaje de programación, donde cada símbolo

en el código fuente de un programa está asociado con información tal como la ubicación, el tipo de datos y el ámbito de cada variable, constante o procedimiento.

## GRAMÁTICAS DE ATRIBUTOS

Las gramáticas de atributos sirven para describir formalmente lenguajes. Una gramática de atributos permite describir:

- a. La sintaxis del lenguaje: utiliza una gramática libre de contexto. Dicha gramática identifica la estructura sintáctica de las frases del lenguaje. Esta estructura puede interpretarse por medio de reglas sintácticas o producciones.
- b. La semántica del lenguaje: añade atributos semánticos a los símbolos de la gramática que se ha creado para la sintaxis del documento y ecuaciones semánticas a las producciones para expresar las acciones a realizar.

Las gramáticas de atributos permiten representar, aparte de la sintaxis de los lenguajes, también la forma en la que las frases de dichos lenguajes son procesadas.

En nuestro proyecto, las gramáticas de atributos serán utilizadas para describir documentos JSON y para especificar las acciones que debe realizar sobre él en el programa que queremos generar.

## CUP

Es un generador de traductores ascendentes para la programación en Java. Trabaja sobre gramáticas LALR.

CUP [12] genera código Java que implementa el analizador sintáctico a partir de un fichero con la especificación sintáctica del lenguaje.

CUP genera dos ficheros, con los nombres por defecto:

1. **Sym.java**: contiene las definiciones de constantes de la clase sym, que asigna un valor entero a cada terminal, y opcionalmente, a cada no terminal.

Ejemplo:

```
//-----  
// The following code was generated by CUP v0.10k  
// Sun Mar 11 01:33:31 CET 2007  
//-----  
/** CUP generated class containing symbol constants. */  
public class sym {  
    /* terminals */  
    public static final int RPARENT = 10;  
    public static final int DIGITO = 6;  
    public static final int POTENCIA = 7;  
    public static final int SUMA = 2;  
    public static final int MULTIPLICACION = 5;  
    public static final int EOF = 0;  
    public static final int SIGNO = 11;  
    public static final int RESULTADO = 8;  
    public static final int error = 1;  
    public static final int DIVISION = 4;  
    public static final int RESTA = 3;  
    public static final int LPARENT = 9;  
}
```

2. **Parser.java**: clase pública del analizador sintáctico.

Contiene dos clases:

- a) public class **parser** extends java\_cup.runtime.lr\_parser{ ...}

Clase pública del analizador.

Es subclase de java\_cup.runtime.lr\_parser que implementa la tabla de acciones de un analizador LALR.

- b) class **CUP\$parser\$actions**

Clase privada incluida en el fichero que encapsula las acciones de usuario contenidas en la gramática.

Contiene el método que selecciona y ejecuta las diferentes acciones definidas en cada regla sintáctica:

```
public final java_cup.runtime.Symbol CUP$parser$do_action
```

## *ESTRUCTURA DE UN FICHERO CUP*

```
import ...;

action code { : ... : }
parser code { : ... : }
init with { : ... : }
scan with { : ... : }

terminal ...;
non terminal ...;

precedence ...;

gramática
```

Un fichero CUP podemos dividirlo en cinco partes, cada una de las cuáles tiene una función en el proceso.

1. Especificaciones de importación y empaquetamiento. La primera parte del fichero se utiliza para incluir la información que define como se generará el analizador y el código de importación de la librería CUP necesario para poder desarrollar el código Java.
2. Código de usuario. Después de las especificaciones de importación y empaquetamiento, se pueden ubicar una serie de declaraciones opcionales que permiten incluir código en el analizador generado.

Action code: código que contiene métodos auxiliares y variables empleados por el código incrustado en la gramática, este código se incrusta en una clase embebida en el parser.

Parser code: código que flexibiliza el uso del parser, este código se incrusta directamente en la clase parser.

Init with: el parser ejecutará el código aquí introducido antes de pedir el primer token. Inicializaciones, instanciaciones...

Scan with: código que devolverá símbolos.

3. Lista de símbolos. La segunda parte de la especificación declara los terminales y no terminales utilizados en la gramática, y opcionalmente, especifica el tipo (clase-objeto) de cada uno de ellos. Si se omite el tipo, al terminal/no terminal correspondiente no se le podrán asignar valores.

4. Declaraciones de precedencia. Define la precedencia y asociatividad de los terminales utilizados en la gramática para evitar ambigüedades.
5. Gramática. La última parte del fichero contiene la gramática del lenguaje. No se utiliza ningún delimitador o separador de secciones, sino que se sitúan secuencialmente en el fichero de especificación sintáctica.

Se define la especificación sintáctica y se incluyen los atributos y las acciones semánticas que permiten manejar los símbolos leídos y realizar las acciones oportunas (comprobaciones, inserciones) sobre la tabla de símbolos

Nota: Estas 5 partes (algunas opcionales) deben introducirse en el orden en el que se han definido.





## JAVACC

JavaCC [4] es una herramienta de generación automática de analizadores gramaticales basada en Java. La herramienta es propiedad de Sun Microsystems, la compañía propietaria del lenguaje Java, por lo que se ha convertido en el metacompilador más usado por los programadores en Java.

Genera un parser para una gramática representada en notación BNF y genera analizadores descendentes, lo que limita a la clase de gramáticas LL(K).

Un generador de analizadores sintácticos es una herramienta que lee la especificación de una gramática y la convierte en un programa java capaz de reconocer coincidencias con la gramática.

El funcionamiento de la herramienta consiste en analizar un fichero de entrada, que contiene la descripción de una gramática, y generar un conjunto de ficheros de salida, escritos en Java, que contienen la especificación de un analizador léxico y de un analizador sintáctico para la gramática especificada.

Las características más importantes de esta herramienta son las siguientes:

- Es la herramienta más utilizada en Java. Los propietarios estiman en cientos de miles el número de descargas de la herramienta y los foros de discusión congregan a miles de usuarios interesados en JavaCC [5].
- Se basa en un análisis sintáctico descendente recursivo.
- Por defecto JavaCC [4] analiza gramáticas de tipo LL(1), pero permite fijar un Lookahead mayor (para analizar gramáticas LL(k)) e incluso utilizar un Lookahead adaptativo.
- Las especificaciones léxica y sintáctica de la gramática a analizar se incluyen en un mismo fichero.
- La especificación léxica se basa en expresiones regulares y la especificación sintáctica utiliza el formato EBNF.
- Junto a la herramienta principal se incluyen dos utilidades: JJTree, para crear automáticamente un generador de árboles sintácticos, y JJDoc, para generar automáticamente la documentación de la gramática en formato HTML.
- La distribución incluye numerosos ejemplos de gramáticas y existen repositorios en internet con la especificación de muchísimas gramáticas en el formato de JavaCC [5].

- La gestión de errores léxicos y sintácticos está basada en excepciones y contiene información muy valiosa respecto al origen del error y su posición.
- Existe un plugin para Eclipse [6] que facilita la edición y ejecución de la herramienta dentro del desarrollo de cualquier aplicación en Java.

## LENGUAJES DE MARCADO

Un lenguaje de marcado es una forma de codificar un documento dentro de una estructura de etiquetas que consigue clasificar los distintos datos que éste contiene.

El lenguaje de marcado más conocido en la actualidad es el HTML. Este lenguaje es utilizado para la codificación de páginas Web, sus etiquetas se encuentran delimitadas por corchetes angulares (<>).

Podemos diferenciar los lenguajes de marcado en tres tipos diferenciados:

### *MARCADO DE PRESENTACIÓN*

Indica el formato del texto. Este tipo es útil para presentar un documento y prepararlo para su visualización y lectura.

Es cada vez menos usado, debido a su complejidad a la hora de mantener el código y su imposibilidad de codificar estructuras de datos, aunque es fácil de escribir para documentos pequeños.

### *MARCADO DE PROCEDIMIENTOS*

Está enfocado hacia la presentación de los textos. Incluye los tamaños de fuente, negrita, centrar párrafos, etc. Estas instrucciones deben aparecer al principio de cada texto para poder ser interpretadas.

Ejemplos de este tipo de marcado son nroff, troff y TEX.

### *MARCADO DESCRIPTIVO*

El marcado descriptivo utiliza etiquetas para clasificar los fragmentos del texto, pero no especifica cómo tratar estos fragmentos, ni lo que representa cada etiqueta.

La principal característica de estos lenguajes es su flexibilidad a la hora de representar los datos. Podemos representar cualquier tipo de metadatos, y basta con que el receptor de los mismos conozca nuestro sistema de marcado para poder representarlos correctamente.

De este modo este tipo de marcado no está limitado por las especificaciones de un lenguaje, sino que es completamente flexible para poder escribir lo que se necesite.

El principal ejemplo de estos lenguajes es el XML, y también se incluye en ellos el JSON [1].

## JSON

JSON [1] es un formato ligero de intercambio de datos, independiente del lenguaje de programación. Tiene forma de texto plano, de simple lectura, escritura y generación. Y además ocupa menos espacio que el formato XML.

Debido a la simplicidad, no es necesario que se construyan parsers personalizados.

Características principales:

- 1) Independiente de un lenguaje específico.
- 2) Basado en texto.
- 3) De formato ligero.
- 4) Fácil de parsear.
- 5) No define funciones.
- 6) No tiene estructuras invisibles.
- 7) No tiene espacios de nombres.
- 8) No tiene validador.
- 9) No es extensible.

Sirve para representar objetos en el lado del cliente, normalmente en aplicaciones RIA que utilizan JavaScript.

### *DOCUMENTO JSON*

```
object::
    { members }
    {}
members ::
    string : value
    members, string : value

array ::
    [ elements ]
    []
elements ::
    value
    elements, value
value ::
    string
    number
    object
    array
    true
    false
    null
```

- ✓ **Object:** conjunto desordenado de pares nombre/valor.  
Conjunto de propiedades, cada una con su valor.  
Empieza con una llave de apertura, y termina con una llave de cierre.  
Propiedades:
  - Se separan con comas
  - El nombre y el valor están separados por dos puntos
- ✓ **Array:** colección ordenada de valores.  
Colección ordenada de valores u objetos.  
Empieza con un corchete izquierdo y acaba con un corchete derecho. Los valores se separan con comas.
- ✓ **Value:** puede ser un string, número, booleano, objeto o array.  
Puede ser una cadena de caracteres con comillas dobles, un número, true, false, null, un objeto o un array.
- ✓ **String:** colección de cero o más caracteres UNICODE encerrados por comillas dobles. Los caracteres de escape utilizan la barra invertida.
- ✓
- ✓ **Number:** valor numérico sin comillas.  
Puede representar: Integer, Real, Scientific.

### *COMPARACIÓN DE JSON CON XML Y OTROS LENGUAJES DE MARCADO*

Es frecuente encontrar aplicaciones que utilizan JSON [1] en vez de XML debido a sus ventajas con respecto a él. Sin embargo, en algunos contextos podemos encontrar convivencia entre JSON [1] y XML, como es el caso de la aplicación de cliente de Google Maps con datos meteorológicos en SOAP, que ha necesitado integrar JSON [1] y XML .

A continuación se muestra un ejemplo simple de definición de barra de menús usando JSON [1] y XML.

JSON:

```
{ "menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      { "value": "New", "onclick": "CreateNewDoc()" },
      { "value": "Open", "onclick": "OpenDoc()" },
      { "value": "Close", "onclick": "CloseDoc()" }
    ]
  }
}
```

### XML:

```
<menu id="file" value="File">
  <popup>
    <menuitem value="New" onclick="CreateNewDoc()" />
    <menuitem value="Open" onclick="OpenDoc()" />
    <menuitem value="Close" onclick="CloseDoc()" />
  </popup>
</menu>
```

Podemos observar en este ejemplo que el espacio ocupado por el código en XML es similar al utilizado por JSON [1], por lo tanto el tamaño no es un motivo para elegir JSON [1] con respecto a XML.

Sin embargo, sí podemos decir que JSON [1] representa de mejor manera los datos en su estructura, siendo más intuitivo en su codificación.

Existen en internet muchas más herramientas de desarrollo para XML que para JSON [1], lo que dificulta el trabajo para un desarrollador que opte por JSON [1] como lenguaje.

Pueden encontrarse más comparaciones entre JSON [1] y XML en la página oficial de JSON [1].

YAML es un superconjunto de JSON [1] que intenta superar algunas de sus limitaciones. Es un lenguaje de marcado más complejo que JSON [1], aunque aún puede considerarse como ligero. El lenguaje de programación Ruby utiliza YAML como formato de deserialización por defecto.

## OBJETIVO DEL PROYECTO

El objetivo de este trabajo de Sistemas Informáticos es la construcción de un entorno de desarrollo que permita entender la construcción de aplicaciones JSON [1] como la construcción de procesadores de lenguaje. Más concretamente, el entorno deberá permitir:

- Especificar cada aplicación de procesamiento JSON [1] como un procesador para el lenguaje de marcado definido por su gramática documental.
- Generar automáticamente el procesador a partir de dicha especificación.

El entorno se denominará JLOP, y ofrecerá un lenguaje de especificación basado en gramáticas de atributos.

En el entorno JLOP será posible:

- Especificar las aplicaciones de procesamiento JSON [1] como gramáticas de atributos.
- Generar automáticamente implementaciones Java de dichas aplicaciones a partir de dichas especificaciones.

Además, en este trabajo también se pretende adquirir conocimientos complementarios a los ya obtenidos a lo largo de los estudios de Ingeniería en Informática relativos a las tecnologías de procesamiento, así como a las herramientas de construcción de procesadores de lenguaje.





## METAGRAMÁTICA JLOP

Para poder representar una correcta gramática que defina un documento JSON [1], hemos tenido que especificar una “metagramática” que sea capaz de contener todas las posibles especificaciones de un documento JSON [1].

Es obvio que existen muchas maneras de codificar estas especificaciones, pero nosotros hemos optado por las siguientes:

```
JLOP ::= Reglas "@"

Reglas ::= Regla Reglas
Reglas ::= λ
Regla ::= Antecedente "::=" Consecuentes "(" Argumentos ")"

Antecedente ::= VAR

Consecuentes ::= Consecuente Consecuentes
Consecuentes ::= λ
Consecuente ::= Antecedente
Consecuente ::= "{" AtriJSON}"
Consecuente ::= "[" AtriJSON]"
Consecuente ::= "," Consecuente

AtriJSON ::= VAR ":" AtriJSON
AtriJSON ::= VALOR
AtriJSON ::= "{" AtriJSON (RAtriJSON)*}"
AtriJSON ::= "[" AtriJSON (RAtriJSON)*]"
AtriJSON ::= "*" Consecuente (RAtriJSON)*
RAtriJSON ::= "," AtriJSON

Argumentos ::= Argumento ";" Argumentos
Argumentos ::= λ
Argumento ::= Atributo "=" Expresion

Expresion ::= Funcion "(" (Expresion)* ")"
Expresion ::= CONSTANT
Expresion ::= Atributo

Atributo ::= "#" VAR "of" Atributo
```

Para representar un atributo referido a una aparición de nuestro lenguaje lo haremos de la siguiente manera:

```
"#" < VAR > " of " token2=< VAR >
```

Es nuestro programa el que se encargará de, al compilarlo, comprobar que estas variables que estamos escribiendo, que podrían ser cualquier cadena de caracteres, coincidan con una aparición de una variable en la producción en la que la estamos referenciando.

Cuando dentro de un atributo JSON [1] necesitamos referenciar a una producción de nuestro lenguaje usaremos el carácter \* de la siguiente manera:

```
"*" (Consecuente()) (RAtriJSON()) *
```

Esto es necesario para evitar bucles con la producción de Consecuente.

Los tokens utilizados para representar las variables son los siguientes:

**TOKEN :**

```
{  
    < CONSTANT : (< DIGIT >)+ >  
| < #DIGIT : [ "0"-"9" ] >  
}
```

**TOKEN :** /\*PARA LOS VALORES DE LOS ATRIBUTOS DENTRO DEL JSON\*/

```
{  
    < VALOR : "INTEGER" | "STRING" >  
}
```

**TOKEN :** /\*PARA LOS ANTECEDENTES/CONSECUENTES\*/

```
{  
    < VAR : (< LETRA >)+ (< NUM > | < LETRA >)* >  
| < #NUM : [ "0"-"9" ] >  
| < #LETRA : [ "A"-"Z", "a"-"z", "_" ] >  
}
```

De este modo, una variable debe comenzar siempre por una letra, y puede ir seguido de una combinación de uno o más números y letras.

El carácter para indicar que hemos finalizado de escribir nuestra gramática será la arroba (@).

Esta metagramática ha sido implementada en la clase JLOP.jj del paquete JLOP.

### Restricciones

Además de esta metagramática que nos asegura unas restricciones mínimas para poder reconocer nuestra gramática de entrada y darla por válida, necesitamos unas restricciones que no nos es posible codificar en la metagramática, por lo que son codificadas en nuestro código jlop.jj, trabajando con los atributos y métodos disponibles en almacen.java.

La primera de estas restricciones es que una vez referenciada una producción, debemos siempre dar un valor a todos sus atributos en cada aparición de ella en la gramática.

Para ello, en su primera aparición, la almacenaremos en una estructura de datos, junto con todos sus atributos. En las sucesivas apariciones comprobaremos que todos sus atributos han sido referenciados correctamente, de lo contrario se mostrará un mensaje de error por la consola.

Otra restricción es que para hacer una correcta interpretación de la producción:

```
Atributo ::= "#" VAR "of" Atributo
```

Es evidente que en la ecuación en la que nos encontremos deberá aparecer la referencia al segundo atributo, de lo contrario no podremos acceder a él.

Por lo demás, somos libres de especificar nuestra gramática como queramos y necesitemos, siempre que nos ajustemos a las restricciones marcadas por nuestra metagramática.



# MODELO DE PROCESO

## CICLO DEL PROYECTO



En cada iteración hemos tenido en cuenta:

- Los objetivos: necesidad que debe cubrir el producto a desarrollar.
- Las alternativas de conseguir los objetivos descritos con éxito desde distintos puntos de vista.
  - Características: requisitos a cumplir.
  - Formas de gestión del sistema.
  - Riesgo asumido con cada alternativa.
- Desarrollar y verificar el software: si el resultado no es el adecuado, se necesitarán implementar mejoras o nuevas funcionalidades.
- Se planificarán los siguientes pasos y se comienza de nuevo el ciclo.

En la espiral generada por el desarrollo del ciclo se pueden observar dos dimensiones:

- Angular: indica el avance del proyecto dentro de un ciclo.
- Radial: indica el aumento del coste del proyecto a medida que pasa el tiempo.

Para cada ciclo identificamos cuatro actividades principales:

1. Determinar objetivos y fijar los productos definidos a obtener: requisitos, especificación, manuales de usuario. Se fijan las restricciones impuestas. Se identifican los riesgos que pueden surgir, y las posibles formas de resolverlos. Se realiza una única vez una planificación inicial.
2. Análisis del riesgo, de las posibles amenazas y las consecuencias que se puedan producir.
3. Planificación: se evalúa el trabajo realizado, se decide si se continúa con el resto de las fases y se planifica la siguiente actividad.
4. Desarrollar y probar. Dependiendo de la evaluación, se elige un modelo para el desarrollo (formal, evolutivo, cascada, etc.)

A la hora de realizar el desarrollo de nuestro proyecto, hemos decidido hacerlo de una manera incremental e iterativa.

Para ello, al comienzo definimos unos objetivos principales, que son los que debíamos satisfacer en primera instancia, y que formarían la base de nuestro proyecto. A medida que el tiempo transcurría, refinábamos esos objetivos, a la vez que incorporábamos nueva funcionalidad al proyecto.

La primera etapa del proyecto consistió en la investigación del entorno en el que se enmarcaría nuestro proyecto, así como de las herramientas que íbamos a utilizar en el desarrollo del mismo.

En esta etapa también elaboramos la metagramática para reconocer la gramática de los documentos JSON [1].

En la segunda etapa, con la ayuda de JavaCC [5], hemos realizado el análisis sintáctico del documento de entrada de la aplicación.

En la tercera etapa, hemos incluido el análisis semántico y visualización final del objetivo del documento sirviéndonos de la herramienta CUP [12].

Hemos buscado ejemplos de documentos JSON [1] para poder establecer unas acciones a realizar por nuestra aplicación.

Por último, hemos realizado una interfaz gráfica para facilitar la interacción de la aplicación con el usuario.

## ARQUITECTURA DEL SISTEMA

Este proyecto ha sido implementado en dos paquetes: JLOP y Proyecto1.

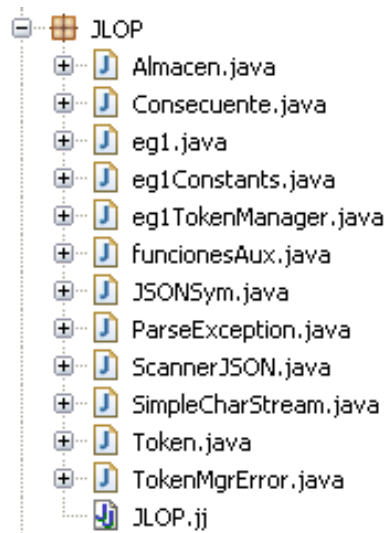


Figura 1

En el paquete JLOP podremos encontrar la clase JLOP.jj. Aquí es donde se ha codificado toda la metagramática en JavaCC.

```
void Consecuente() :
{
{
Antecedente()
|
"{" AtriJSON() (RAtriJSON()) * "}"
|
"[" AtriJSON() (RAtriJSON()) * "]"
|
"," Consecuente()
}
}

void RAtriJSON() :
{
{
"," AtriJSON()
}
}

void AtriJSON() :
{
{
< VAR > ":" AtriJSON()
|
< VALOR >
|
"{" AtriJSON() (RAtriJSON()) * "}"
|
"[" AtriJSON() (RAtriJSON()) * "]"
|
"*" (Consecuente()) (RAtriJSON()) *
}
}
```

Figura 2

Además, en esta clase, se han implementado todas las restricciones adicionales a la metagramática, necesarias para la correcta compilación de las gramáticas de entrada.

Esto es necesario debido a la imposibilidad de representar algunas restricciones dentro de nuestra metagramática.

Esto se ha implementado en conjunción con la clase `almacen.java`, que contiene todos los métodos necesarios para la contención de la información, y que veremos más tarde.

Este es un ejemplo de restricción fuera de la metagramática:

```
void Atributo() :
{
    Token token2;
}
{
    "##" < VAR > " of " token2=< VAR >
    {
        System.out.println("Debe aparecer en la ecuación la referencia a " + token2);
        if (almacen1.existeCons(token2)) System.out.println("Existe, OK");
        else if (almacen1.getTokenActual().next.next.toString()=="{")
            System.out.println(token2+" es una referencia a un atributo JSON");
        else System.out.println("ERROR, no existe referencia a "+token2+" en la ecuación");
    }
    |
    < VAR >
}
```

**Figura 3**

Aquí se puede observar que en esta producción de `Atributo`, hace una llamada a la clase `almacen` sobre el `token2`, el cual será contrastado con la base de datos, para saber si ha aparecido en la ecuación en la que nos encontramos, o por el contrario, es una mala referencia y debe ser marcado como un error de la gramática.

Todas las restricciones, así como la especificación completa de la metagramática pueden consultarse en el Anexo1 Metagramática JLOP.



Finalmente, al compilar nuestro archivo JLOP.jj se generará otro archivo eg1.java, que contiene su especificación generada en código java para poder ejecutarla.

```
static final public void RatriJSON() throws ParseException {
    jj_consume_token(30);
    AtriJSON();
}

static final public void AtriJSON() throws ParseException {
    switch ((jj_ntk==-1)?jj_ntk():jj_ntk) {
    case VAR:
        jj_consume_token(VAR);
        jj_consume_token(31);
        AtriJSON();
        break;
    case VALOR:
        jj_consume_token(VALOR);
        break;
    case 26:
        jj_consume_token(26);
        AtriJSON();
        label_5:
        while (true) {
            switch ((jj_ntk==-1)?jj_ntk():jj_ntk) {
            case 30:
                ;
                break;
            default:
                jj_la1[6] = jj_gen;
                break label_5;
            }
        }
        RAtriJSON();
    }
}
```

Figura 4

La clase más importante y el eje de nuestra estructurar de datos es la clase almacen.java.

Esta clase contiene todos los tokens y la información importante leída de la gramática que se ha introducido.

Para ello utilizamos varias estructuras de datos como arrays y punteros a los mismos para poder conocer siempre nuestra posición en ellos.

En la siguiente figura podemos ver los atributos que contiene la clase almacen, y en los que vamos a almacenar toda la información necesaria tanto para el reconocimiento de la gramática como para la posterior escritura de un documento CUP a partir de ella.

```

public class Almacen {

    private Token[][] matriz;
    private Token[][] matrizReglas;
    private Integer[] punterosArg;
    private Integer[] punterosReglas;
    private Token[] terminales;
    private Token[] atriJSON;
    private Token tokenActual;
    private Token[] argumentosActual;
    private Token[] consecuentesActual;
    private Integer numConsecuentesActual;
    private Integer puntero;
    private Integer punteroReglas;
    private Integer punteroTerminales;
    private Integer punteroJSON;
    private boolean primero;
    private boolean esAntecedente;
    private File fichero;
    private PrintWriter bw;
}

```

Figura 5

Esta clase tiene no sólo tiene la funcionalidad de ir recibiendo los datos de entrada, sino que al finalizar la entrada, se llama a uno de sus métodos, que se encarga de recorrer todas las estructuras para generar un fichero Parser.cup, en lenguaje cup a partir de la gramática de entrada.

Esta es otra de las partes importantes de este proyecto, ya que se trata de que un programa en JavaCC [5], a través de una clase escrita en Java, genere automáticamente un programa en Cup [12]:

```

import java.io.*;
import java_cup.*;
import java_cup.runtime.*;
import JLOP.ScannerJSON;
import JLOP.Consecuente;
import JLOP.funcionesAux;
/*init with {
parser analizador;
analizador = new parser(new ScannerJSON());
:};*/

scan with {
    return getScanner().next_token();
    :};

parser code {
    public static void main (String argv[]) throws Exception
    {
        File file=new File("C:/JSON/JSON.txt");
        try{

            FileReader fr=new FileReader(file);
            Scanner lex=new ScannerJSON(fr);
            ParserCup miParser=new ParserCup(lex);
            miParser.parse();

        }catch(Exception e){
            System.out.println(e);
        }
    }
:};

```

**Figura 6**

Esta es la cabecera del archivo Parser.cup, es importante porque recoge la manera de recorrer el archivo JSON, su localización y nombre, y las librerías necesarias.

Además el archivo cup debe contener los terminales y los no terminales de la gramática:

```
// terminales y no terminales
terminal Attribute abre;
terminal Attribute dospuntos;
terminal Attribute cierra;
terminal Attribute coma;
terminal Attribute abreCorch;
terminal Attribute cierraCorch;
terminal Consecuente CONS;
terminal String STRING;
terminal Integer INTEGER;
non terminal prog;
non terminal Resultados;
non terminal Resultados2;
non terminal Resultado;
terminal fin;
terminal results;
terminal address_components;
terminal formatted_address;
terminal geometry;
terminal bounds;
terminal northeast;
terminal lat;
terminal lng;
terminal southwest;
terminal location;
terminal location_type;
terminal viewport;
terminal types;
terminal status;
terminal long_name;
terminal short_name;
```

Figura 7

Y por supuesto también debe especificar la gramática en sí:

```
// gramatica
prog ::= abre CONS:E1 dospuntos abreCorch abre CONS:E2 dospuntos Resultados coma CONS:E3 dospuntos CONS:E4 coma C
{ :
RESULT= new Consecuente();
Consecuente anterior = (Consecuente) ((java_cup.runtime.Symbol) CUP$ParserCup$stack.elementAt (CUP$ParserCup$top-1))
:}
;

Resultados ::= abreCorch Resultados2 cierraCorch
{ :
RESULT= new Consecuente();
Consecuente anterior = (Consecuente) ((java_cup.runtime.Symbol) CUP$ParserCup$stack.elementAt (CUP$ParserCup$top-1))
:}
;

Resultados2 ::= Resultados2 coma Resultado
{ :
RESULT= new Consecuente();
Consecuente anterior = (Consecuente) ((java_cup.runtime.Symbol) CUP$ParserCup$stack.elementAt (CUP$ParserCup$top-1))
:}
;

Resultados2 ::= Resultado
{ :
RESULT= new Consecuente();
Consecuente anterior = (Consecuente) ((java_cup.runtime.Symbol) CUP$ParserCup$stack.elementAt (CUP$ParserCup$top-1))
:}
;

Resultado ::= abre CONS:E1 dospuntos CONS:E2 coma CONS:E3 dospuntos CONS:E4 coma CONS:E5 dospuntos abreCorch CONS
{ :
RESULT= new Consecuente();
Consecuente anterior = (Consecuente) ((java_cup.runtime.Symbol) CUP$ParserCup$stack.elementAt (CUP$ParserCup$top-1))
Consecuente fin=funcionesAux.IMPRIME (E2 , E4 , E6) ;
```

Figura 8

En esta captura de código se puede ver la codificación de la gramática. Es importante conocer bien la estructura en forma de pila que se utilizará al recorrer los tokens del archivo JSON, para poder referenciar correctamente a los atributos.

Se puede apreciar también la llama a la función auxiliar “IMPRIME”, y como está contenida dentro de la clase funcionesAux.java.

En funcionesAux.java podemos implementar cualquier método que necesitemos para tratar nuestros datos en formato JSON:

```
public class funcionesAux {
    public static Consecuente SUMA(String a1, String a2)
    {
        return new Consecuente("valor", Integer.toString(Integer.valueOf(a1)+Integer.valueOf(a2)));
    }

    public static Consecuente ESCRIBE(String a1)
    {
        System.out.println(a1);
        return new Consecuente("valor",a1);
    }

    public static Consecuente IMPRIME(Consecuente e1, Consecuente e2, Consecuente e3) {
        System.out.println("Nombre largo: "+e1.getAtributo("valor"));
        System.out.println("Nombre corto: "+e2.getAtributo("valor"));
        System.out.println("Tipo : "+e3.getAtributo("valor"));
        System.out.println(" ");
        return new Consecuente("valor","OK");
    }
}
```

Figura 9

Como se puede observar, es importante nuestra clase Consecuente, que refiere a un token de la gramática.

```
public class Consecuente {
    private String[][] listaAtributos;
    private Integer puntero;
    public Consecuente()
    {
        listaAtributos= new String[10][2];
        puntero=1;
        for (int i=0; i<10; i++)
        {
            for (int j=0; j<2; j++)
            {
                listaAtributos[i][j]= "XXX";
            }
        }
    }
    public Consecuente(String atributo, String valor) {
        listaAtributos= new String[10][2];
        puntero=1;
        for (int i=0; i<10; i++)
        {
            for (int j=0; j<2; j++)
            {
                listaAtributos[i][j]= "XXX";
            }
        }
        listaAtributos[1][0]=atributo;
        listaAtributos[1][1]=valor;
        puntero++;
    }
}
```

Figura 10

En esta clase recogemos los posibles atributos que tiene un consecuente. Esto es importante a la hora de comprobar la coherencia de la gramática, ya que se deben referenciar siempre todos los atributos de un consecuente en una producción.

Para recorrer los archivos JSON [1] en busca de tokens, utilizamos la clase ScannerJSON.java:

```
public Symbol next_token() throws Exception {
    // TODO Auto-generated method stub
    JSONSym s = new JSONSym();
    //int num;
    comillas=false;
    while (true) {
        switch(leido) {
            case '\\0': {
                return new Symbol(s.EOF,new Consecuente("valor","EOF"));
            }
            case '{': {
                leido = (char) entrada.read();
                return new Symbol(s.abre,new Consecuente("valor","{"));
            }
            case '}': {
                leido = (char) entrada.read();
                return new Symbol(s.cierra,new Consecuente("valor","}"));
            }
            case ':': {
                leido = (char) entrada.read();
                return new Symbol(s.dospuntos,new Consecuente("valor",":"));
            }
            case ',': {
                leido = (char) entrada.read();
                return new Symbol(s.coma,new Consecuente("valor",","));
            }
            case '[': {
                leido = (char) entrada.read();
                return new Symbol(s.abreCorch,new Consecuente("valor","["));
            }
            case ']': {
                leido = (char) entrada.read();
                return new Symbol(s.cierraCorch,new Consecuente("valor","]"));
            }
        }
    }
}
```

Figura 11

Es importante implementar correctamente el método next\_token(), ya que será llamado por el cup cada vez que necesite un token nuevo, y éste debe ser devuelto en forma de Symbol de la manera esperada.

Se construye un symbol a partir de una pareja de un integer y un objeto. El integer coincide con el tipo de símbolo que estamos enviando, recogido en la clase ParserSym.java. El objeto es en nuestro caso un consecuente, que contiene toda la información que necesitamos sobre el token en cuestión, para poder tratarlo correctamente.



Además, el Parser.cup generará un Parsercup.java y un ParserSym.java, en el paquete proyecto1:

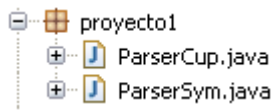


Figura 12

El ParserCup.java codifica el código a ejecutar para cada posible producción de la gramática.

Podemos observar como codifica cada producción por separado y almacena en la variable RESULT el resultado de cada producción para irlo propagando por el árbol:

```
RESULT= new Consecuente();
Consecuente anterior = (Consecuente) ((java_cup.runtime.Symbol) CUP$ParserCup$stack.elementAt (CUP$ParserCup$top-1
    CUP$ParserCup$result = parser.getSymbolFactory().newSymbol("Resultados2",2, ((java_cup.runtime.S{
    }
    return CUP$ParserCup$result;

/* . . . . . */
case 3: // Resultados2 ::= Resultados2 coma Resultado
{
    Object RESULT =null;

RESULT= new Consecuente();
Consecuente anterior = (Consecuente) ((java_cup.runtime.Symbol) CUP$ParserCup$stack.elementAt (CUP$ParserCup$top-1
    CUP$ParserCup$result = parser.getSymbolFactory().newSymbol("Resultados2",2, ((java_cup.runtime.S{
    }
    return CUP$ParserCup$result;

/* . . . . . */
case 2: // Resultados ::= abreCorch Resultados2 cierraCorch
{
    Object RESULT =null;

RESULT= new Consecuente();
Consecuente anterior = (Consecuente) ((java_cup.runtime.Symbol) CUP$ParserCup$stack.elementAt (CUP$ParserCup$top-1
    CUP$ParserCup$result = parser.getSymbolFactory().newSymbol("Resultados",1, ((java_cup.runtime.Syn
    }
    return CUP$ParserCup$result;

/* . . . . . */
case 1: // $START ::= prog EOF
{
    Object RESULT =null;
```

Figura 13

Todos los terminales son codificados por el ParserSym.java:

```
/** CUP generated interface containing symbol constants. */
public interface ParserSym {
    /* terminals */
    public static final int short_name = 27;
    public static final int location = 21;
    public static final int coma = 5;
    public static final int geometry = 15;
    public static final int lat = 18;
    public static final int abre = 2;
    public static final int bounds = 16;
    public static final int long_name = 26;
    public static final int CONS = 8;
    public static final int cierraCorch = 7;
    public static final int types = 24;
    public static final int STRING = 9;
    public static final int formatted_address = 14;
    public static final int EOF = 0;
    public static final int address_components = 13;
    public static final int results = 12;
    public static final int status = 25;
    public static final int cierra = 4;
    public static final int error = 1;
    public static final int northeast = 17;
    public static final int location_type = 22;
    public static final int fin = 11;
    public static final int INTEGER = 10;
    public static final int viewport = 23;
    public static final int southwest = 20;
    public static final int dospuntos = 3;
    public static final int lng = 19;
    public static final int abreCorch = 6;
}
```

Figura 14

Con estos atributos, podremos referenciar más tarde desde el parser de nuestro archivo JSON [1] cada elemento que nos vayamos encontrando, clasificándolo como integer, string o cualquiera de las etiquetas que contiene el tipo de JSON [1] especificado en nuestra gramática de entrada.

## CASO DE ESTUDIO

Para nuestro caso de estudio, hemos elegido procesar archivos JSON [1] provenientes de una consulta realizada al motor de búsqueda de Google Maps.

Estos archivos tienen un formato específico, que representamos con la gramática:

```
prog ::=
{
  results : [
    {
      address_components :
      * Resultados
    ,
    formatted_address : STRING,
    geometry : {
      bounds : {
        northeast : {
          lat : STRING , lng : STRING
        },
        southwest : {
          lat : STRING , lng : STRING
        }
      },
      location :
      * Location
    ,
    location_type : STRING,
    viewport : {
      northeast : {
        lat : STRING , lng : STRING
      },
      southwest : {
        lat : STRING , lng : STRING
      }
    }
  },
  types : [ STRING, STRING ]
}
],
status : STRING
}
(
)
```

```
Resultados ::= [ * Resultados2 ]
(
)
```

```
Resultados2 ::= Resultados2 , Resultado
(
)
```

```
Resultados2 ::= Resultado
(
)
```

```

Resultado ::= {
    long_name : STRING,
    short_name : STRING,
    types : [ STRING, STRING ]
}

(
fin = IMPRIME(long_name short_name types);
)

Location ::= {
    lat : STRING ,
    lng : STRING
}

(
fin2 = IMPRIME (lat lng);
)

@

```

Como función aplicable a estos documentos hemos utilizado dos funciones, aunque de igual nombre: IMPRIMIR.

En el caso de los resultados, llamamos a la función imprimir con 3 argumentos, y nos devuelve los resultados de la búsqueda por pantalla.

En el caso de la latitud, la llamaremos con 2 argumentos, y pintará sobre un mapa la localización del resultado.

Para obtener un documento JSON [1] de Google Maps, debemos enviarle una petición del tipo:

<http://maps.googleapis.com/maps/api/geocode/json?address=DIRECCION&sensor=false>

Donde "DIRECCION" se refiere a la cadena que queremos buscar.

El ejemplo que se mostrará proviene de realizar la búsqueda sobre la cadena “Madrid”, que devuelve los siguientes datos:

```
{
  "results" : [
    {
      "address_components" : [
        {
          "long_name" : "Madrid",
          "short_name" : "Madrid",
          "types" : [ "locality", "political" ]
        },
        {
          "long_name" : "Madrid",
          "short_name" : "M",
          "types" : [ "administrative_area_level_2", "political" ]
        },
        {
          "long_name" : "Comunidad de Madrid",
          "short_name" : "Comunidad de Madrid",
          "types" : [ "administrative_area_level_1", "political" ]
        },
        {
          "long_name" : "España",
          "short_name" : "ES",
          "types" : [ "country", "political" ]
        }
      ],
      "formatted_address" : "Madrid, España",
      "geometry" : {
        "bounds" : {
          "northeast" : {
            "lat" : 40.64351810,
            "lng" : -3.51801020
          },
          "southwest" : {
            "lat" : 40.31207130,
            "lng" : -3.88900490
          }
        },
        "location" : {
          "lat" : 40.41669090,
          "lng" : -3.7003453999999999
        },
        "location_type" : "APPROXIMATE",
        "viewport" : {
          "northeast" : {
            "lat" : 40.64351810,
            "lng" : -3.51801020
          },
          "southwest" : {
            "lat" : 40.31207130,
            "lng" : -3.88900490
          }
        }
      },
      "types" : [ "locality", "political" ]
    }
  ],
  "status" : "OK"
}
```

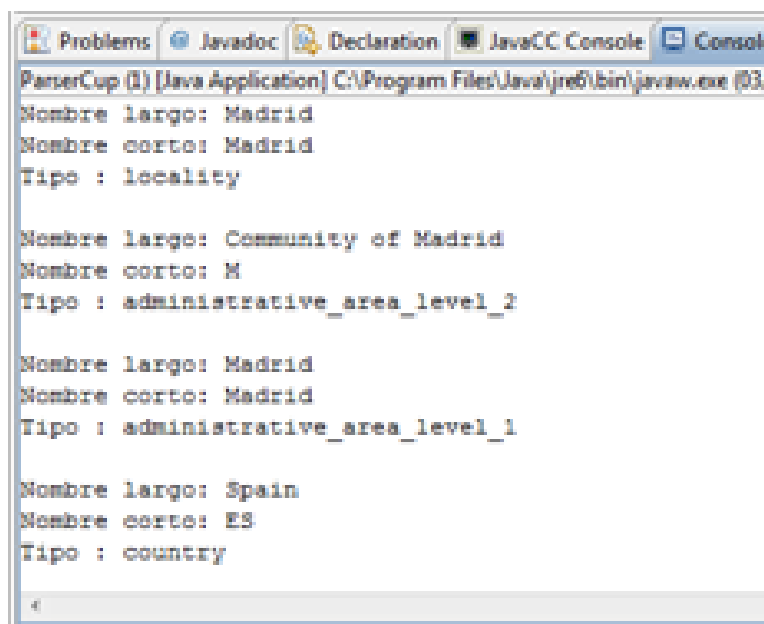
Observando estos resultados, podemos ver que la estructura devuelta por Google Maps no es más que una lista de lo que hemos llamado Resultados, que contienen información sobre su ubicación.

La primera parte de la información es su nombre, corto y largo, que hemos utilizado para nuestra función aplicada a la gramática, la función IMPRIME:

La función IMPRIME con 3 argumentos está codificada de la siguiente manera:

```
public static Consecuente IMPRIME(Consecuente e1, Consecuente
e2, Consecuente e3) {
    System.out.println("Nombre largo:
"+e1.getAtributo("valor"));
    System.out.println("Nombre corto:
"+e2.getAtributo("valor"));
    System.out.println("Tipo : "+e3.getAtributo("valor"));
    System.out.println(" ");
    return new Consecuente("valor", "OK");
}
```

Ejemplo de ejecución:



```
Pantecup (1) [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (03)
Nombre largo: Madrid
Nombre corto: Madrid
Tipo : locality

Nombre largo: Community of Madrid
Nombre corto: M
Tipo : administrative_area_level_2

Nombre largo: Madrid
Nombre corto: Madrid
Tipo : administrative_area_level_1

Nombre largo: Spain
Nombre corto: ES
Tipo : country
```

Figura 15

Esta es una función realmente sencilla que lo único que realiza es una impresión por pantalla de los atributos del documento JSON [1].

La segunda parte de la información contenida en el resultado de la búsqueda en Google Maps son las coordenadas exactas de la localización del resultado.

La función IMPRIME con 2 argumentos es la siguiente:

```
public static Consecuente IMPRIME(Consecuente e1, Consecuente e2) {  
    System.out.println("ESTO SON LAS COORDENADAS DEL MAPA");  
    new map(  
        Double.valueOf(e1.getAtributo("valor")).doubleValue(),  
        Double.valueOf(e2.getAtributo("valor")).doubleValue()  
    );  
    return new Consecuente("valor", "OK");  
}
```

Y como se puede apreciar, utiliza la clase map, que tendrá que estar definida para poder ser invocada.

Esta función es llamada en la gramática sobre los datos de localización del resultado, por lo que contiene la latitud y longitud del mismo.

Este es un ejemplo de lo que genera la clase map que hemos implementado para este ejemplo:



Figura 16

Se puede apreciar, aunque la figura es algo pequeña, un círculo rojo, dónde se cruzan las líneas, que representa la localización en el mapa de nuestro resultado recogido en el archivo JSON.

Estas funciones se encuentran implementadas en la clase funcionesAux.java, dentro del paquete JLOP.

Dentro de esta clase podemos implementar cualquier función que podamos codificar en Java, utilizando todas las herramientas que necesitemos para su ejecución, y nuestro programa se encargará de ejecutarla una vez reciba la correspondiente llamada en el código del lenguaje.

En realidad, cualquier función que podamos escribir en Java puede ser utilizada sobre estos documentos, el objetivo de este proyecto es sólo dar el soporte para poder construir estas funciones, sea cual sea su objetivo.

Esta aplicación de Google Maps es una de las más importantes que utilizan JSON, y esta es la razón de que la hayamos escogido como ejemplo y caso de estudio.

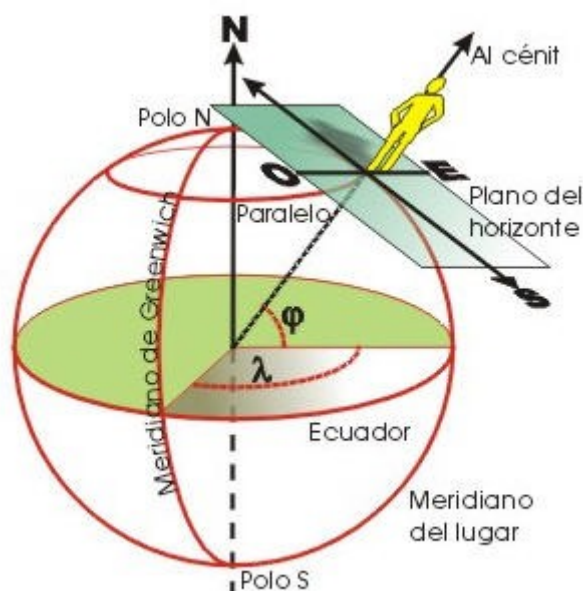


## COORDENADAS GEOGRÁFICAS

Las coordenadas geográficas son un conjunto de líneas imaginarias, o coordenadas angulares que permiten ubicar con exactitud un lugar en la superficie de la Tierra. Este conjunto de líneas corresponden a los meridianos y paralelos.

Cualquier punto de nuestro planeta puede ubicarse al conocerse el meridiano de longitud y el paralelo de latitud.

Estas dos coordenadas angulares medidas desde el centro de la Tierra son de un sistema de coordenadas esféricas que están alineadas con su eje de un sistema de coordenadas geográficas incluye un datum, meridiano principal y unidad angular. Estas coordenadas se suelen expresar en grados sexagesimales.



La latitud mide el ángulo entre cualquier punto y el ecuador. Las líneas de latitud se llaman meridianos y son círculos que cortan al ecuador en la superficie de la Tierra. La latitud es el ángulo que existe entre un punto cualquiera y el Ecuador, medida sobre el meridiano que pasa por dicho punto. La distancia en km a la que equivale un grado depende de la latitud, a medida que la latitud aumenta disminuyen los kilómetros por grado.

La longitud mide el ángulo a lo largo del ecuador desde cualquier punto de la Tierra. Se acepta que Greenwich en Londres es la longitud 0 en la mayoría de las sociedades modernas. Las líneas de longitud son círculos máximos que pasan por los polos y se llaman meridianos.

Combinando estos dos ángulos, se puede expresar la posición de cualquier punto de la superficie de la Tierra. Por ejemplo, Baltimore, Maryland (en los Estados Unidos), tiene latitud 39,3 grados norte, y longitud 76,6 grados oeste. Así un vector dibujado desde el centro de la tierra al punto 39,3 grados norte del ecuador y 76,6 grados al oeste de Greenwich pasará por Baltimore.

El ecuador es un elemento importante de este sistema de coordenadas; representa el cero de los ángulos de latitud y el punto medio entre los polos. Es el plano fundamental del sistema de coordenadas geográficas.

## LOCALIZACIÓN GEOGRÁFICA DE UN PUNTO

La localización geográfica de un punto se puede realizar detallando uno de los siguientes parámetros:

- Coordenadas geográficas en formato Longitud-Latitud.
- Coordenadas (x,y) UTM.

Cada una de estas dos formas de localizar un punto sobre la superficie terrestre debe cumplir los siguientes requisitos:

- Que el punto sea único.
- Que quede perfectamente identificado el sistema de proyección empleado al localizar el punto.
- Que permita referenciar la coordenada "z" del punto.

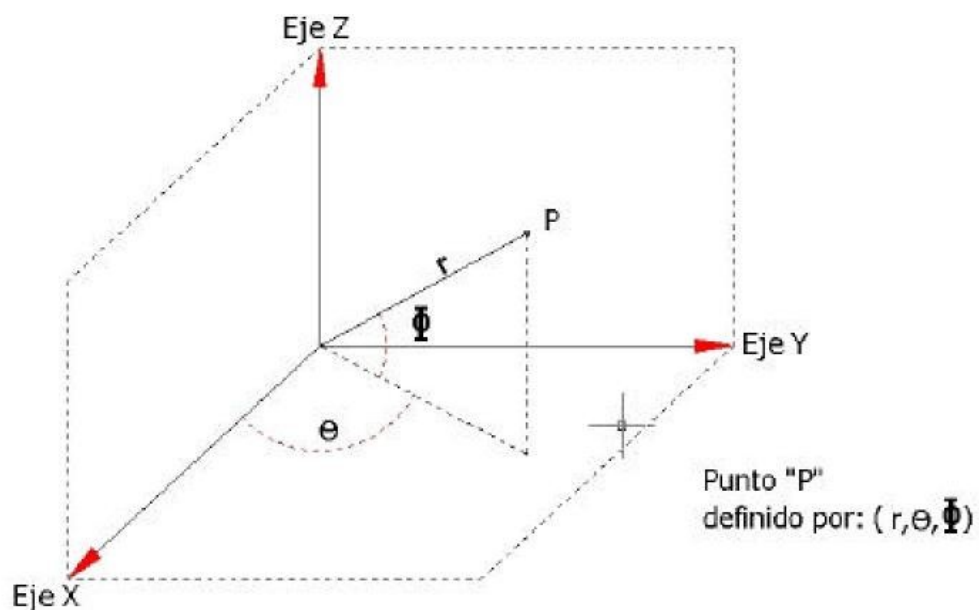
La forma de designar un punto en función de sus coordenadas geográficas se hace de la siguiente manera:

Grados ° Minutos ' Segundos " Orientación

Donde orientación toma los valores según sea Norte (N), Sur (S), Este (E), Oeste (W).

**3°14'26" W**  
**42°52'21" N**

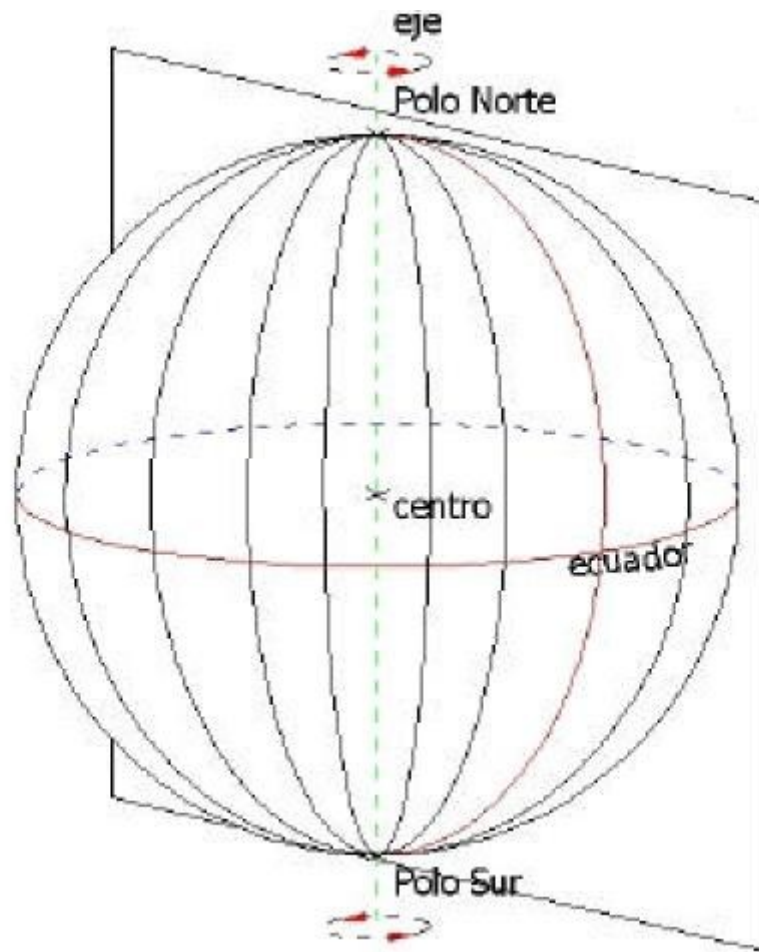
Esta forma de representar las coordenadas, supone la creación de un sistema de referencia de tres dimensiones:



## MERIDIANOS

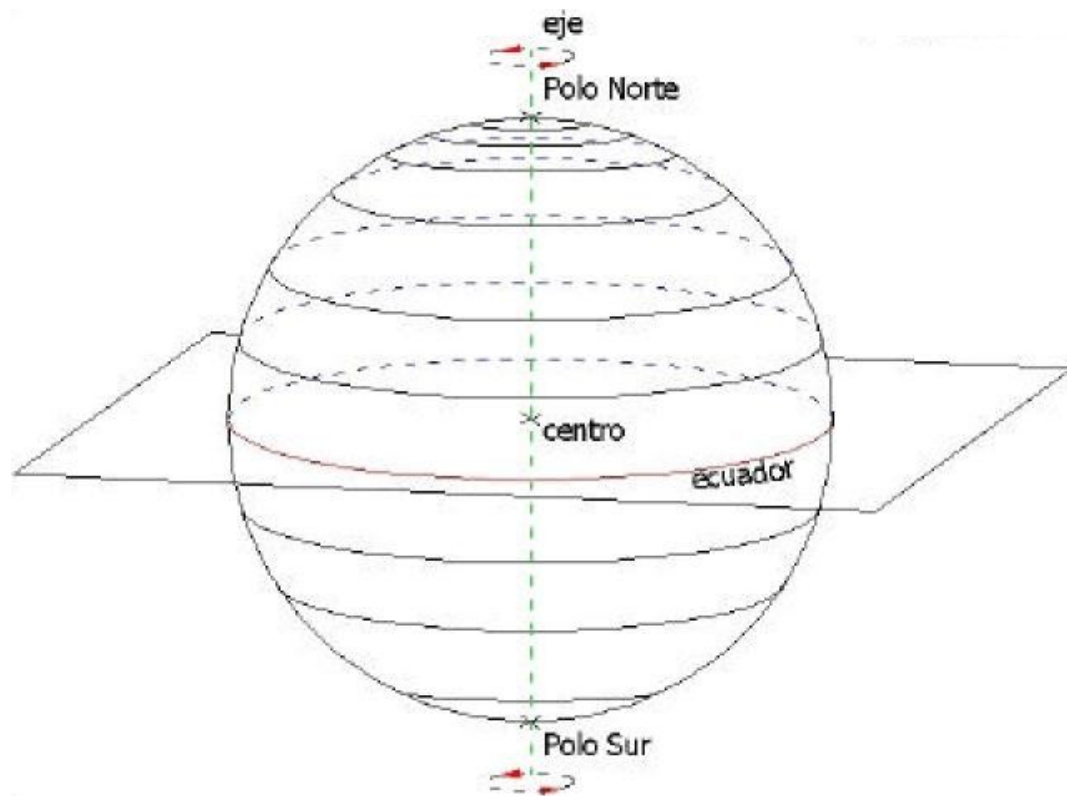
Se definen como las líneas de intersección con la superficie terrestre de los infinitos planos que contienen al eje de la tierra.

Se define el eje de la tierra como la recta ideal de giro del globo terráqueo en su giro del movimiento de rotación.



## PARALELOS

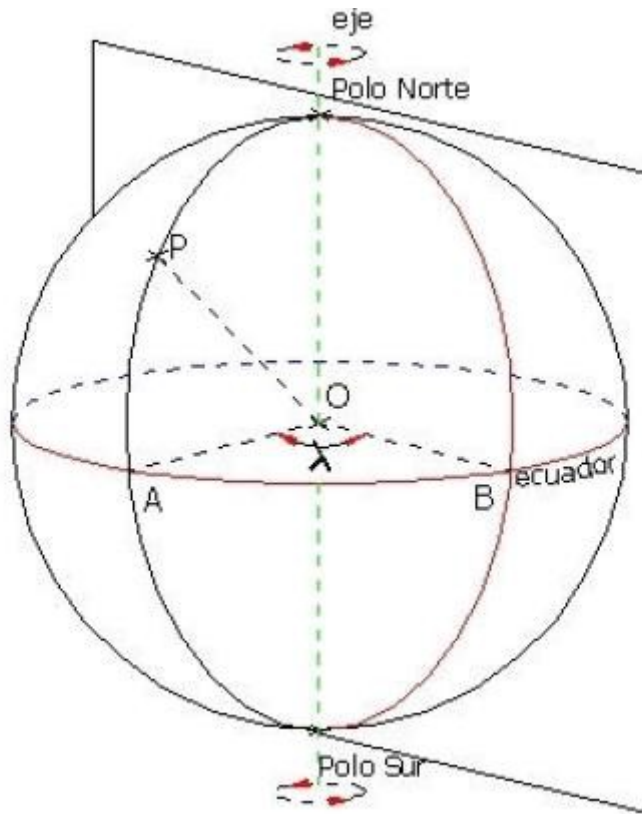
Se definen como la intersección de los infinitos planos perpendiculares al eje terrestre con la superficie de la tierra.



## LONGITUD

La longitud ( $\lambda$ ) de un punto (P) es el valor del diedro formado por el plano meridiano que pasa por P y el meridiano origen.

Gráficamente se trata del ángulo formado por OAB según se muestra en la figura:



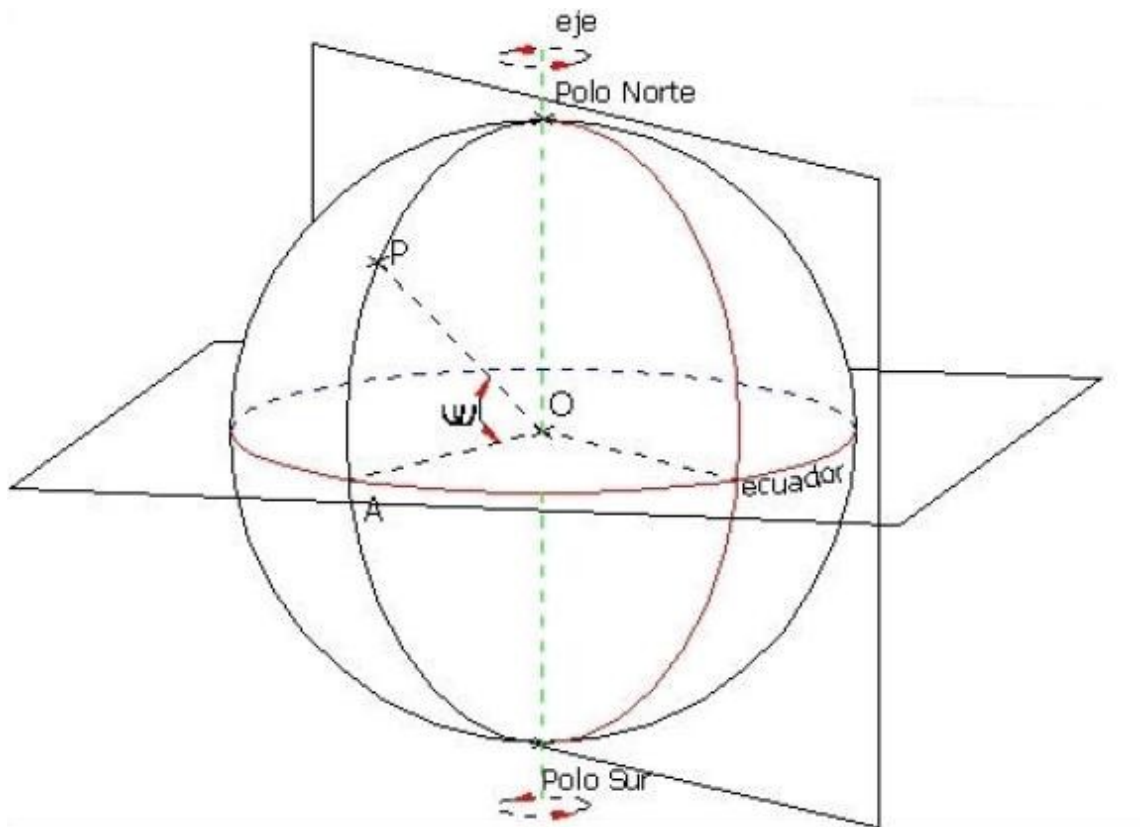
En base a la longitud, se designa la posición del punto, según se encuentre a la izquierda del meridiano origen (Oeste) o a la derecha (Este).

La longitud presenta un mínimo posible de  $0^\circ$  y un máximo de  $180^\circ$

## LATITUD

La latitud geográfica ( $\omega$ ) de un punto (P) es el ángulo formado por la vertical a la tierra que pasa por dicho punto con el plano ecuador.

La vertical se considera la unión del punto con el centro de la tierra, obteniéndose la latitud como el ángulo sobre el meridiano que pasa por el punto.



Los valores para la latitud oscilan desde el mínimo  $0^\circ$  hasta el máximo  $90^\circ$ .





## GOOGLE MAPS

Google Maps es el nombre de un servicio de Google. Es un servidor de aplicaciones de mapas en la Web. Ofrece imágenes de mapas desplazables, así como fotos satelitales del mundo e incluso la ruta entre diferentes ubicaciones o imágenes a pie de calle Google Street View. Desde el 6 de octubre del 2005, Google Maps es parte de Google Local.

Ofrece, asimismo, la posibilidad de que cualquier propietario de una página Web integre muchas de sus características a su sitio.

Con el objetivo de determinar la ubicación de un punto específico dentro de un mapa a gran escala, se puede hacer uso de dos sistemas de coordenadas: métricas y geográficas. Este último permite ubicar un punto cualquiera sobre la superficie terrestre, en relación con una red de paralelos y meridianos. De esta manera, se mide la distancia desde el punto hasta las líneas bases del sistema: El Ecuador (en el caso de la latitud) y Greenwich (en el caso de la longitud), cabe destacar que la ubicación obtenida se expresa en grados sexagesimales.



## CONCLUSIONES

Durante este proyecto hemos aprendido a utilizar toda la potencia que ofrecen los documentos JSON [1] para la codificación de información, consiguiendo completar los objetivos propuestos inicialmente.

Mediante la realización de este trabajo, hemos podido profundizar en nuestros conocimientos de la estructura JSON, los procesadores de lenguajes y gramáticas de atributos, así como de las herramientas disponibles para su construcción, de las cuales hemos hecho uso de JavaCC [5] y CUP [12].

Este proyecto es una base sobre la cual pueden ser construidas infinidad de aplicaciones, con la potencia y versatilidad que quiera darles el programador que las especifique.

A partir de este inicio, queda mucho por desarrollar y muchas posibilidades abiertas para cualquier persona que sepa escribir las necesidades de su programa adecuadas a la metagramática que hemos especificado.



## BIBLIOGRAFÍA

- [1] <http://www.json.org/>
- [2] <http://es.wikipedia.org/wiki/JSON>
- [3] [http://www.slideshare.net/Emmerson\\_Miranda/json-short-manual-5732305](http://www.slideshare.net/Emmerson_Miranda/json-short-manual-5732305)
- [4] <http://es.wikipedia.org/wiki/JavaCC>
- [5] <http://javacc.java.net/>
- [6] <http://eclipse-javacc.sourceforge.net/>
- [7] <http://www.uhu.es/470004004/practicas/practica04.htm>
- [8] <http://www.engr.mun.ca/~theo/JavaCC-Tutorial/javacc-tutorial.pdf>
- [9] <http://www.rose-hulman.edu/class/csse/csse404/201130/Handouts/CUPLEX.html>
- [10] <http://www2.cs.tum.edu/projects/cup/>
- [11] <http://www.cc.uah.es/ie/docencia/ProcesadoresDeLenguaje/CUP.pdf>
- [12] <http://www.cs.princeton.edu/~appel/modern/java/CUP/manual.html>
- [13] <http://crysol.org/es/node/819>
- [14] <http://www.cs.auckland.ac.nz/~bruce-h/lectures/330ChaptersPDF/Chapt4.pdf>
- [15] <http://fibonaccidavinci.wordpress.com/category/mundo-java/>
- [16] <http://manglar.uninorte.edu.co/bitstream/10584/2109/1/1044421326.pdf>
- [17] <http://marcoalopez.blogspot.com.es/2005/11/investigacin-manejo-de-atributos-en.html>
- [18] <http://www.rafaelvega.info/wp-content/uploads/Articulo.pdf>
- [19] <http://code.google.com/p/my-c-compiler/source/browse/trunk/my-gcc/cup/Parser.cup>
- [20] <http://www.slideshare.net/launasa/jlex-cup>
- [21] [http://chuwiki.chuidiang.org/index.php?title=Lectura\\_y\\_Escritura\\_de\\_Ficheros\\_en\\_Java#Ficheros\\_binarios](http://chuwiki.chuidiang.org/index.php?title=Lectura_y_Escritura_de_Ficheros_en_Java#Ficheros_binarios)
- [22] <https://developers.google.com/maps/documentation/localsearch/jsondevguide?hl=es>
- [23] <https://developers.google.com/maps/documentation/geocoding/>
- [24] <http://stackoverflow.com/questions/120283/working-with-latitude-longitude-values-in-java>
- [25] <http://joseguerreroa.wordpress.com/2010/11/01/conversion-de-coordenadas-gauss-kruger-a-geograficas-usando-los-elipsoides-hayford-o-wgs-84-caso-argentina/>
- [26] [http://es.wikipedia.org/wiki/Sistema\\_de\\_Coordenadas\\_Universal\\_Transversal\\_de\\_Mercator](http://es.wikipedia.org/wiki/Sistema_de_Coordenadas_Universal_Transversal_de_Mercator)
- [27] [http://es.wikipedia.org/wiki/Sistema\\_de\\_posicionamiento\\_global](http://es.wikipedia.org/wiki/Sistema_de_posicionamiento_global)
- [28] <http://www.xatakaciencia.com/sabias-que/las-coordenadas-geograficas>
- [29] <http://es.wikipedia.org/wiki/Compilador>



## ANEXO 1. EJEMPLO COMPLETO

Especificación del programa que usaremos como ejemplo para el manual de usuario y para probar la funcionalidad de nuestro trabajo:

```
prog ::=
{
  results : [
    {
      address_components :
      * Resultados
    ,
    formatted_address : STRING,
    geometry : {
      bounds : {
        northeast : {
          lat : STRING , lng : STRING
        },
        southwest : {
          lat : STRING , lng : STRING
        }
      },
      location :
      * Location
    ,
    location_type : STRING,
    viewport : {
      northeast : {
        lat : STRING , lng : STRING
      },
      southwest : {
        lat : STRING , lng : STRING
      }
    }
  },
  types : [ STRING, STRING ]
}
],
status : STRING
}
(
)
```

```
Resultados ::= [ * Resultados2 ]
(
)
```

```
Resultados2 ::= Resultados2 , Resultado
(
)
```

```
Resultados2 ::= Resultado
(
)
```

```

    )

Resultado ::= {
    long_name : STRING,
    short_name : STRING,
    types : [ STRING, STRING ]
}

(
fin = IMPRIME(long_name short_name types);
)

Location ::= {
    lat : STRING ,
    lng : STRING
}

(
fin2 = IMPRIME (lat lng);
)

@

```

Se puede ver que las funciones IMPRIMIR se llaman sobre Resultados y sobre Localización.

La llamada sobre resultados imprimirá por pantalla los textos y títulos y la segunda llamada sobre la localización recogerá la latitud y la longitud y las pintará sobre un mapa utilizando librerías de java.



## ANEXO 2. MANUAL DE USUARIO

Una vez descrita toda la estructura de clases de nuestro proyecto, pasamos a escribir un pequeño manual de usuario, para poder ejecutar de manera correcta nuestro programa.

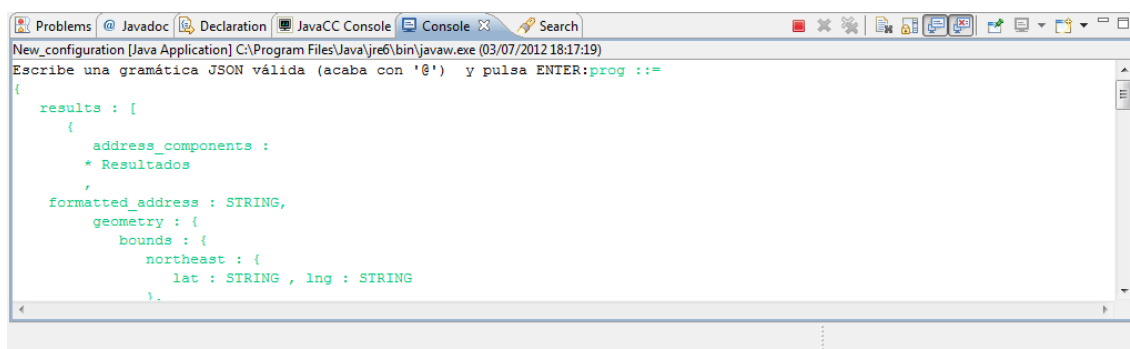
Para su realización hemos tomado como ejemplo nuestro caso de estudio sobre los documentos JSON [1] devueltos por Google Maps tras una consulta anónima a su base de datos.

Una pequeña guía para ejecutar nuestro proyecto es la siguiente:

1. El primer paso es ejecutar el eg1.java y escribir por la consola la gramática que representa los JSON [1] que queremos reconocer e indicar las acciones que se van a hacer con él.

En nuestro caso, la gramática será la tratada en el apartado anterior (Ejemplo Completo), que codifica los documentos JSON [1] devueltos por Google Maps.

Esta es una captura de pantalla de la consola tras pasarle nuestra gramática:



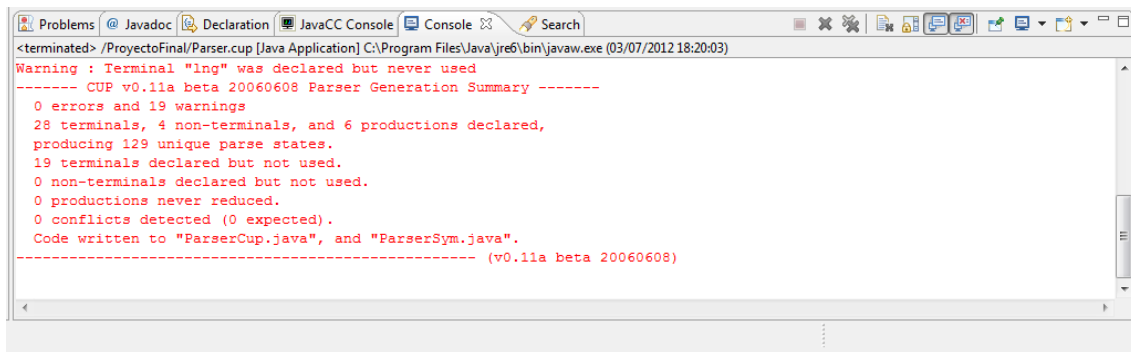
```
Problems | Javadoc | Declaration | JavaCC Console | Console | Search
New_configuration [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (03/07/2012 18:17:19)
Escribe una gramática JSON válida (acaba con '@') y pulsa ENTER:prog ::=
{
    results : [
        {
            address_components :
            * Resultados
        },
        formatted_address : STRING,
        geometry : {
            bounds : {
                northeast : {
                    lat : STRING , lng : STRING
                }
            }
        }
    ]
}
```

Figura 17

Esto generará un archivo Parser.cup, que contendrá la información en CUP [12] de la gramática que hemos insertado.

2. Abrir el Parser.cup. Al compilar este código, se generarán de manera automática dos archivos de Java: ParserCUP.java y ParserSym.java, que contendrán la información que tenía el archivo CUP para codificada en Java para poder ejecutarla.

Esta es una captura de pantalla de la consola tras compilar el archivo Parser.cup:



```
<terminated> /ProyectoFinal/Parser.cup [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (03/07/2012 18:20:03)
Warning : Terminal "lng" was declared but never used
----- CUP v0.11a beta 20060608 Parser Generation Summary -----
0 errors and 19 warnings
28 terminals, 4 non-terminals, and 6 productions declared,
producing 129 unique parse states.
19 terminals declared but not used.
0 non-terminals declared but not used.
0 productions never reduced.
0 conflicts detected (0 expected).
Code written to "ParserCup.java", and "ParserSym.java".
----- (v0.11a beta 20060608)
```

Figura 18

De esta manera podremos acceder a ellos más tarde a través de la interfaz ParserSym.

3. Lo siguiente es crear el JSON [1] que va a ser reconocido por el programa generado, para poder probar su ejecución. Un ejemplo de JSON [1] para probar la gramática:

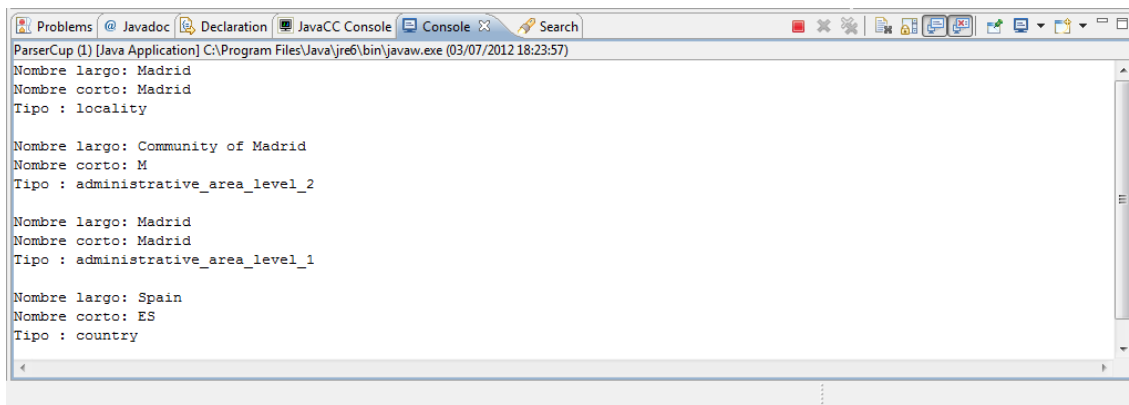
```
{
  "results" : [
    {
      "address_components" : [
        {
          "long_name" : "Madrid",
          "short_name" : "Madrid",
          "types" : [ "locality", "political" ]
        },
        {
          "long_name" : "Community of Madrid",
          "short_name" : "M",
          "types" : [ "administrative_area_level_2", "political" ]
        }
      ],
      {
        "long_name" : "Madrid",
        "short_name" : "Madrid",
        "types" : [ "administrative_area_level_1", "political" ]
      },
      {
        "long_name" : "Spain",
        "short_name" : "ES",
        "types" : [ "country", "political" ]
      }
    ],
    "formatted_address" : "Madrid, Spain",
    "geometry" : {
      "bounds" : {
        "northeast" : {
          "lat" : 40.64351810,
          "lng" : -3.51801020
        },
        "southwest" : {
```

```

        "lat" : 40.31207130,
        "lng" : -3.88900490
    },
    "location" : {
        "lat" : 40.41669090,
        "lng" : -3.7003453999999999
    },
    "location_type" : "APPROXIMATE",
    "viewport" : {
        "northeast" : {
            "lat" : 40.64351810,
            "lng" : -3.51801020
        },
        "southwest" : {
            "lat" : 40.31207130,
            "lng" : -3.88900490
        }
    },
    "types" : [ "locality", "political" ]
},
"status" : "OK"

```

4. Ahora debemos ejecutar el ParserCUP.java. Esto dará como resultado:



**Figura 19**

Esta salida es la implementada por el método IMPRIME, que se encuentra en funcionesAux.java. Cualquier función que se quisiera ejecutar, deberá estar implementada en esa clase.

Como este ejemplo era demasiado sencillo, implementamos una nueva función IMPRIME, dedicada a los atributos de localización, que contienen las latitudes y longitudes de las coordenadas del resultado.

Utilizando librerías de Java, y definiendo una clase mapa, pintamos sobre un JPanel la localización exacta del resultado en un mapamundi.

El resultado es el siguiente:



Si ampliamos la zona de España, que es donde se localiza el resultado que hemos buscado en el ejemplo, podremos apreciar la marca, un círculo rojo situado en Madrid, en el cruce de las líneas perpendiculares, que en este dibujo tan pequeño es más difícil de distinguir:



Con esto finaliza este manual de usuario, dejando el terreno libre para implementar cualquier tipo de documento JSON [1] y cualquier función que deseemos ejecutar sobre el mismo.

## GLOSARIO DE TÉRMINOS Y ACRÓNIMOS

**JSON** JavaScript Object Notation

**JLOP** JSON Language-Oriented Processing

**XML** eXtensible Markup Language

**AJAX** Asynchronous JavaScript And XML

**XSLT** Extensible Stylesheet Language Transformations

**EBNF** Extended Backus–Naur Form

**CUP** Constructor of Useful Parsers

**LALR** Look-Ahead LR parser

**JavaCC** Java Compiler Compiler

**BNF** Backus Normal Form

**LL** LeftMost Derivation

**EBNF** Extended Backus–Naur Form

**HTML** HyperText Markup Language

**RIA** Rich Internet Application

**UTM** Universal Transversa Mercator

**YAML** YAML Ain't Another Markup Language

